ДОДАТОК А

Слайди презентації

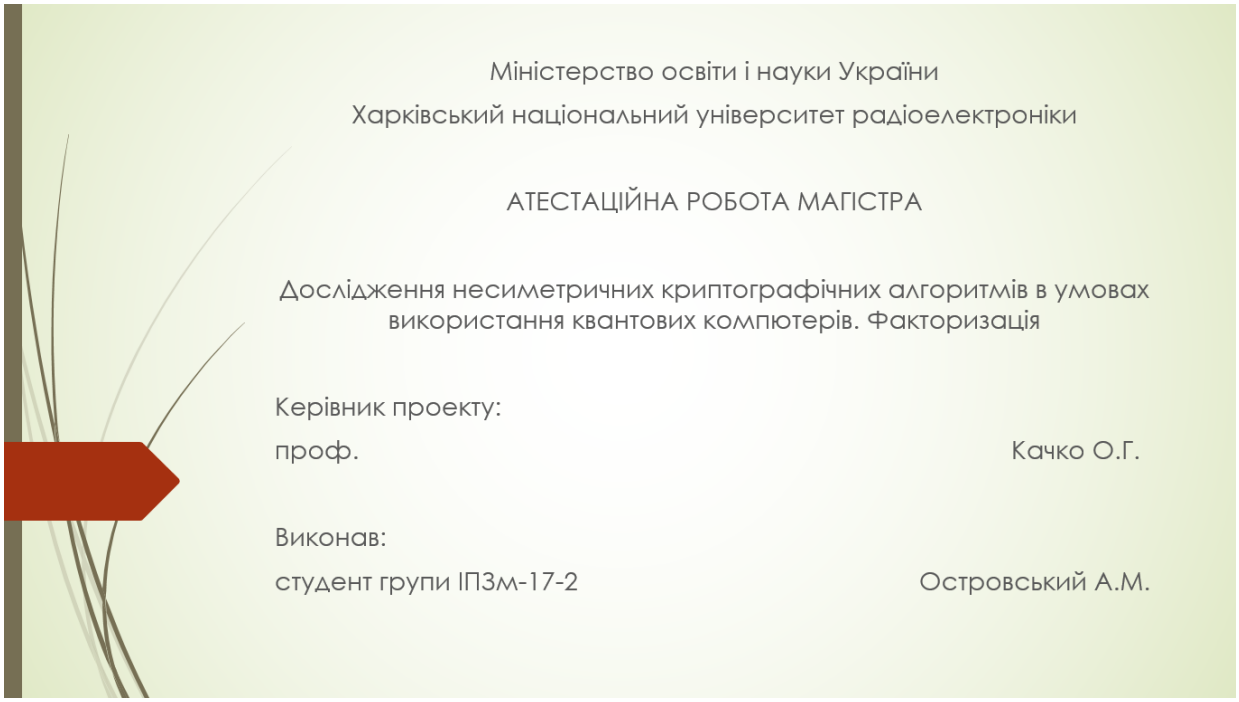

Рисунок А.1 – Слайд презентації №1

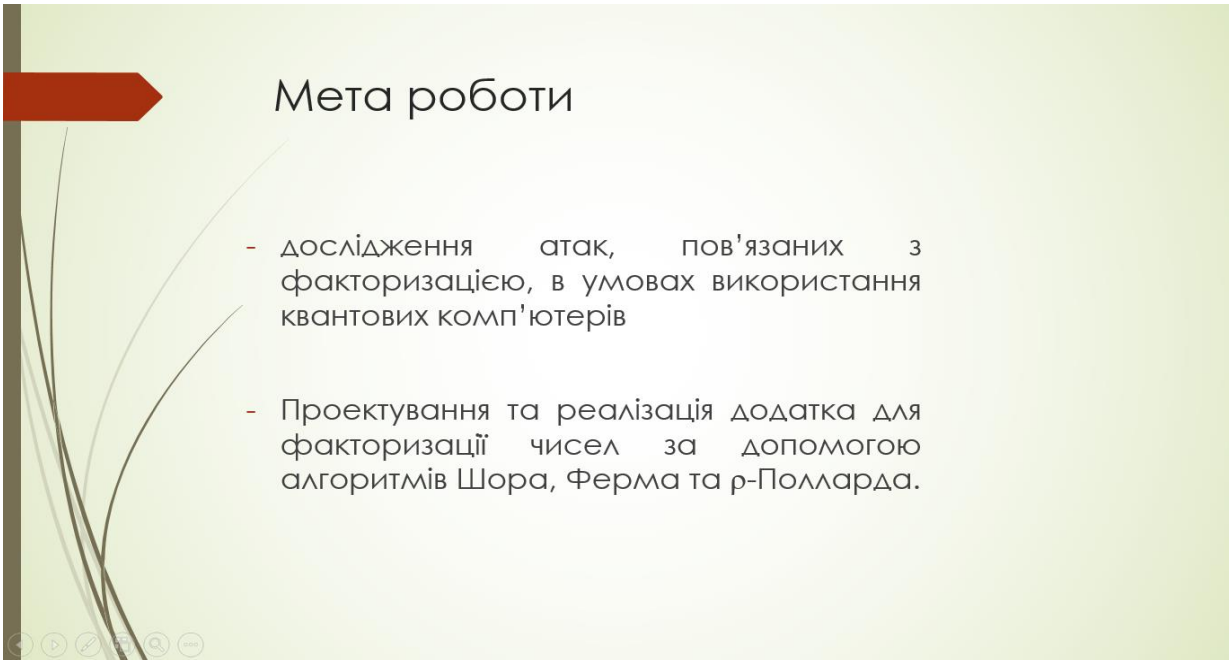

Рисунок А.2 – Слайд презентації №2

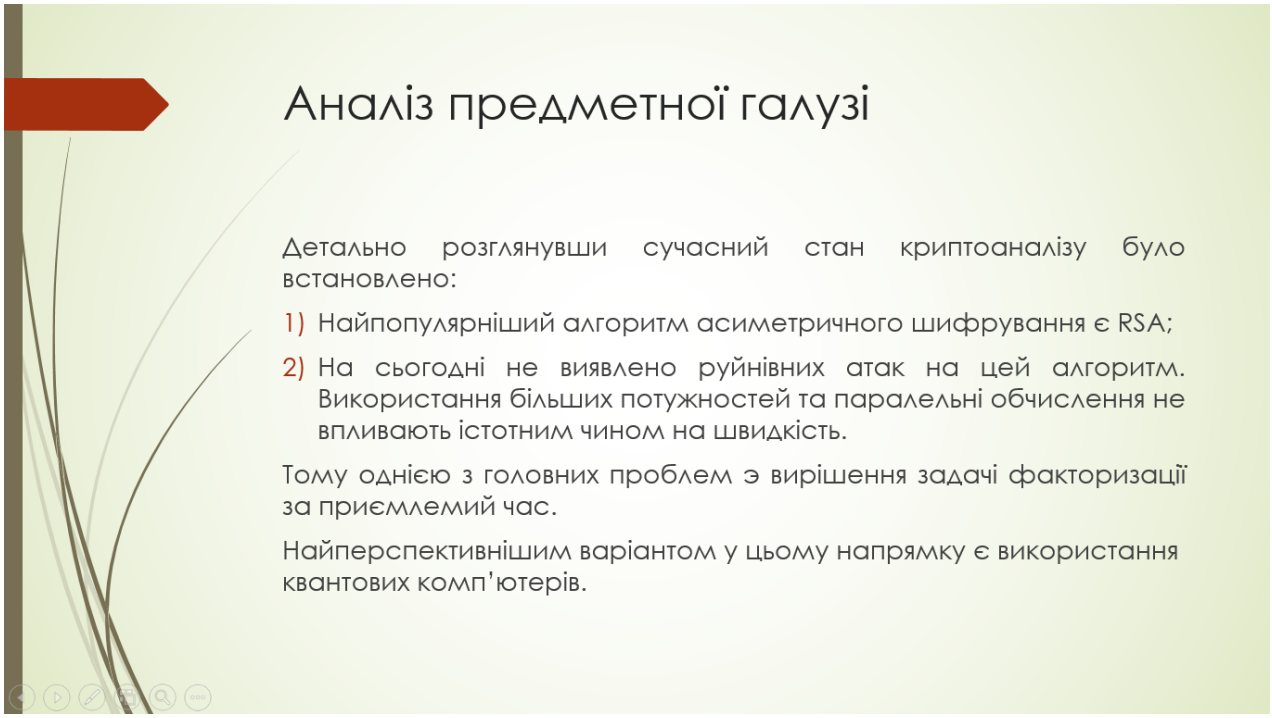

Рисунок А.3 – Слайд презентації №3

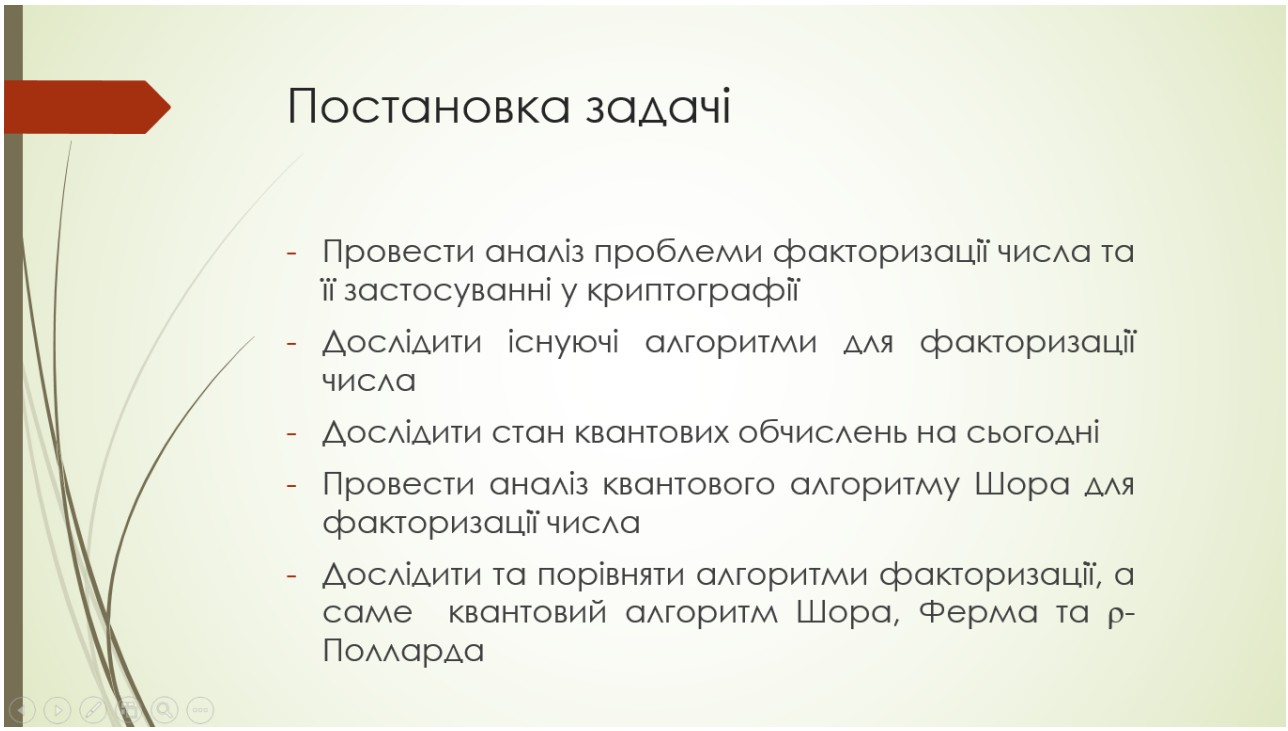

Рисунок А.4 – Слайд презентації №4

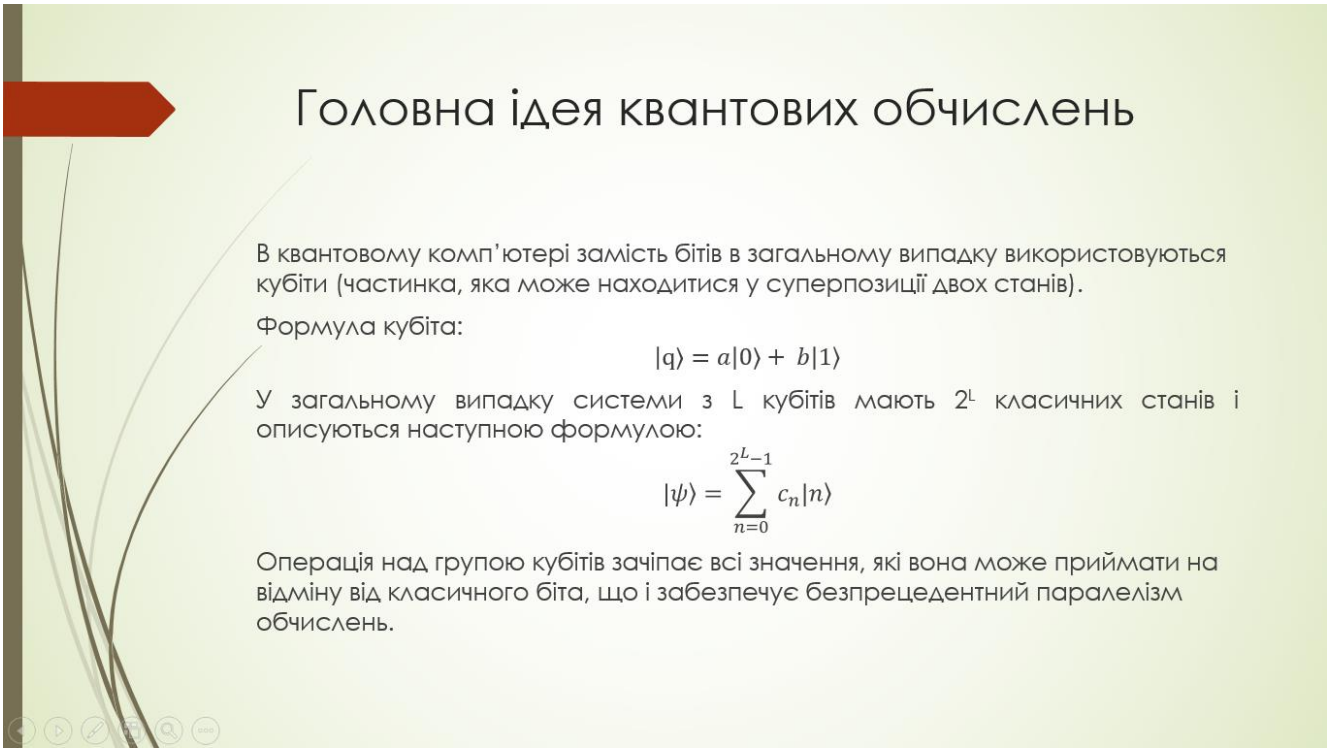

Рисунок А.5 – Слайд презентації №5

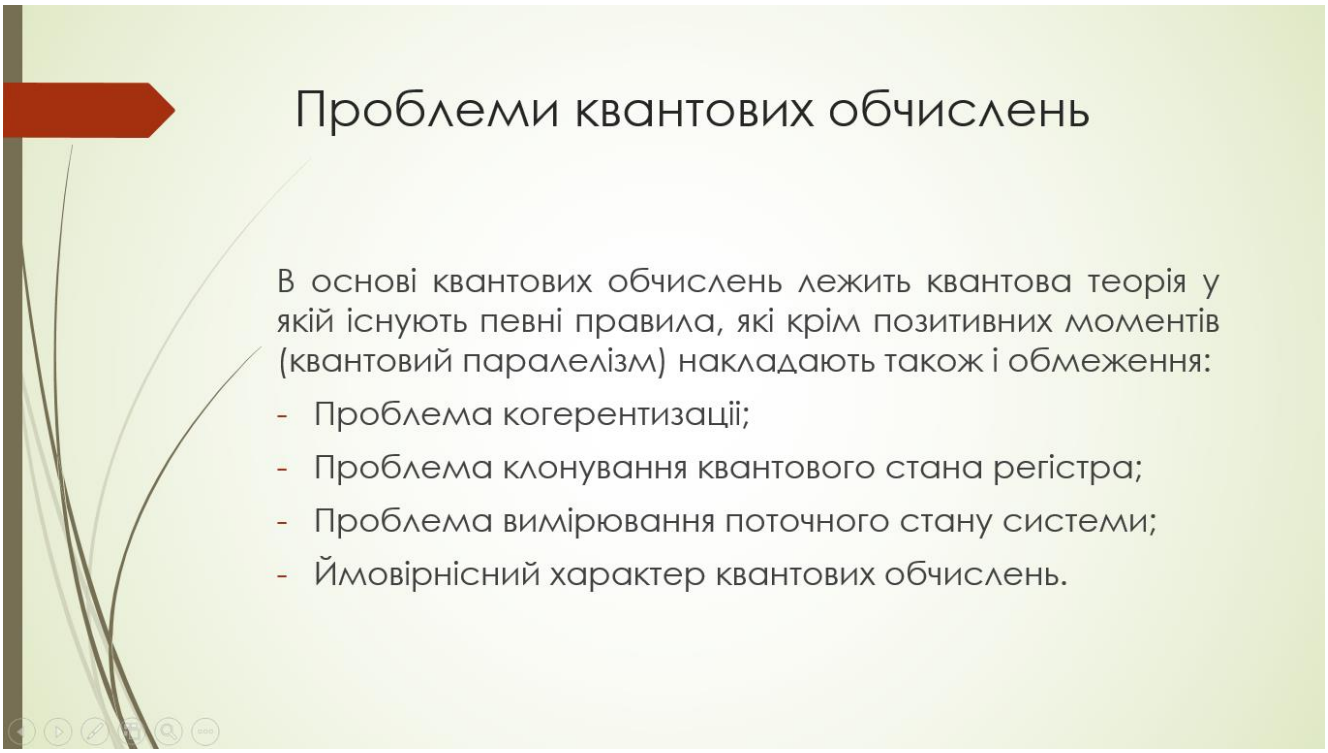

Рисунок А.6 – Слайд презентації №6

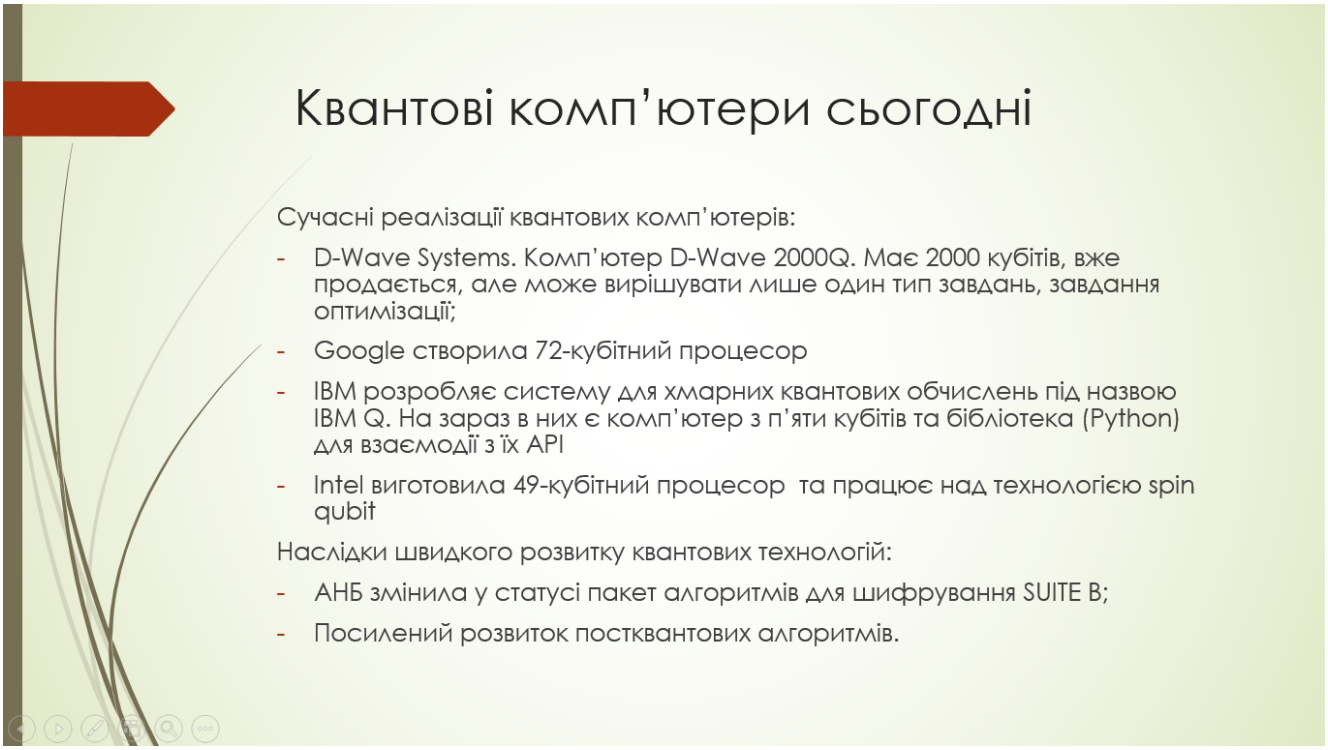

Рисунок А.7 – Слайд презентації №7

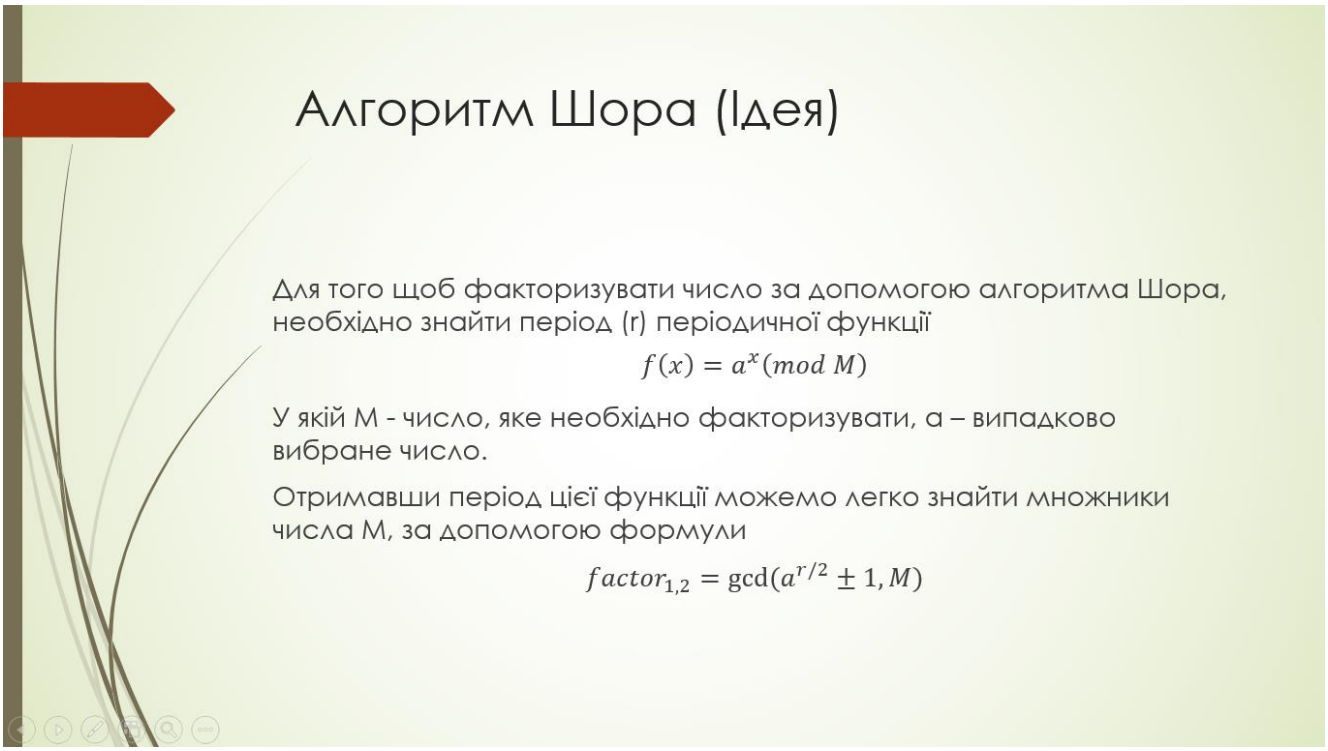

Рисунок А.8 – Слайд презентації №8

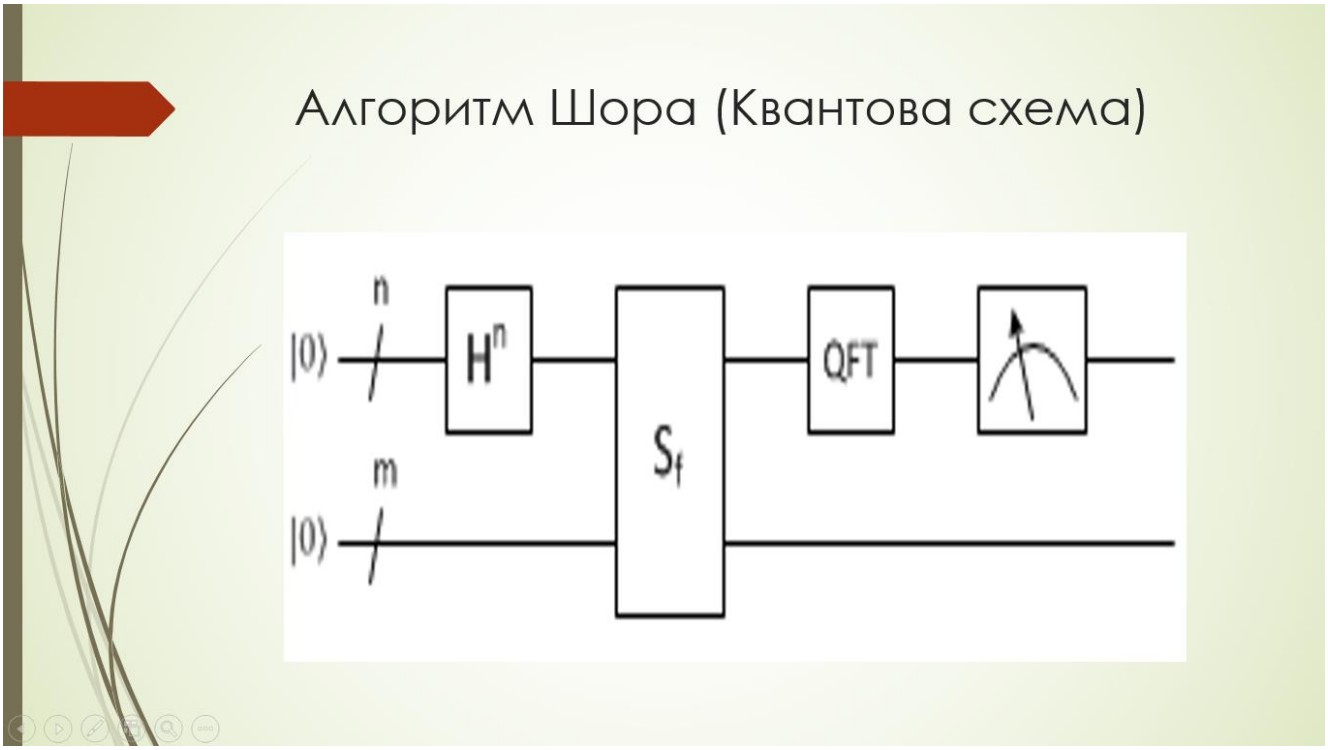

Рисунок А.9 – Слайд презентації №9

Рисунок А.10 – Слайд презентації №10

UML моделювання (Діаграма послідовності)

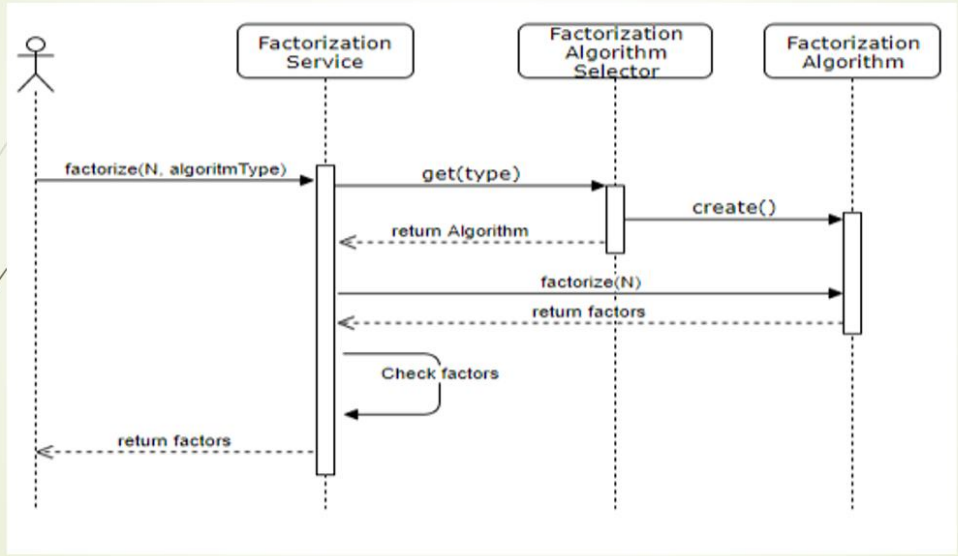

Рисунок А.11 – Слайд презентації №11

Результати виконання додатку

Алгоритм Ферма

№	Число для факторизації	Кількість бітів	Час виконання у послідовному режимі, мс	Час виконання у паралельному режимі, мс
1	3419400499457568584899	72	433	736
2	460675721310754924280 33	76	1912	3013
3	726189210599864879544499	80	251	3237
4	955478587464040811537 2229	83	7435	10673
5	179834419632725580066084077	88	10168	16083
6	299524084095766756626 3635413	92	30476	9350
7	372575586709111509496100 42479	95	81326	41267
8	814916356174804576740926024243	100	108254	210845
9	776929432601662754031471 4751953	103	261663	191769

Алгоритм ρ-Полларда

№	Число для факторизації	Кількість бітів	Час виконання у послідовному режимі, мс	Час виконання у паралельному режимі, мс
1	2326505459	32	30	64
2	46766504593	36	85	101
3	483942319181	39	221	336
4	14722750530251	44	54	76
5	189314357834207	48	4774	3549
6	3390521606369323	52	24581	19926
7	50152109240242930	56	101478	71308
8	885719410306513319	60	2626471	1784782

Класичний алгоритм Шора

№	Число для факторизації	Випадковий параметр (a)	Період функції	Час виконання у послідовному режимі мс.	Час виконання у паралельному режимі мс.
1	360011	2	5980	26	9
2	1434313	2	89440	593	75
3	9173503	2	4583684	5375	2822
4	84698077	2	3528240	4340	1746
5	889310809	2	18525480	48506	22354
6	1439949001	11	143986690	942274	489336

Змодельований алгоритм Шора

Число	Розмір, біт	Номер запуска	Час виконання, мс	Випадковий параметр (a)	Кількість запусків оракула	Середній час, мс
15	4	1	45137	8	2	28137
		2	40462	8	2	
		3	17945	7	1	
		4	20391	7	1	
		5	16753	4	1	
21	5	1	136805	10	4	73878
		2	46294	13	1	
		3	47127	13	1	
		4	92065	10	2	
		5	47101	8	1	
35	6	1	141813	31	1	217344
		2	421745	31	4	
		3	150665	13	1	
		4	146714	29	1	
		5	225783	23	2	

Рисунок А.12 – Слайд презентації №12

Рисунок А.13 – Слайд презентації №13

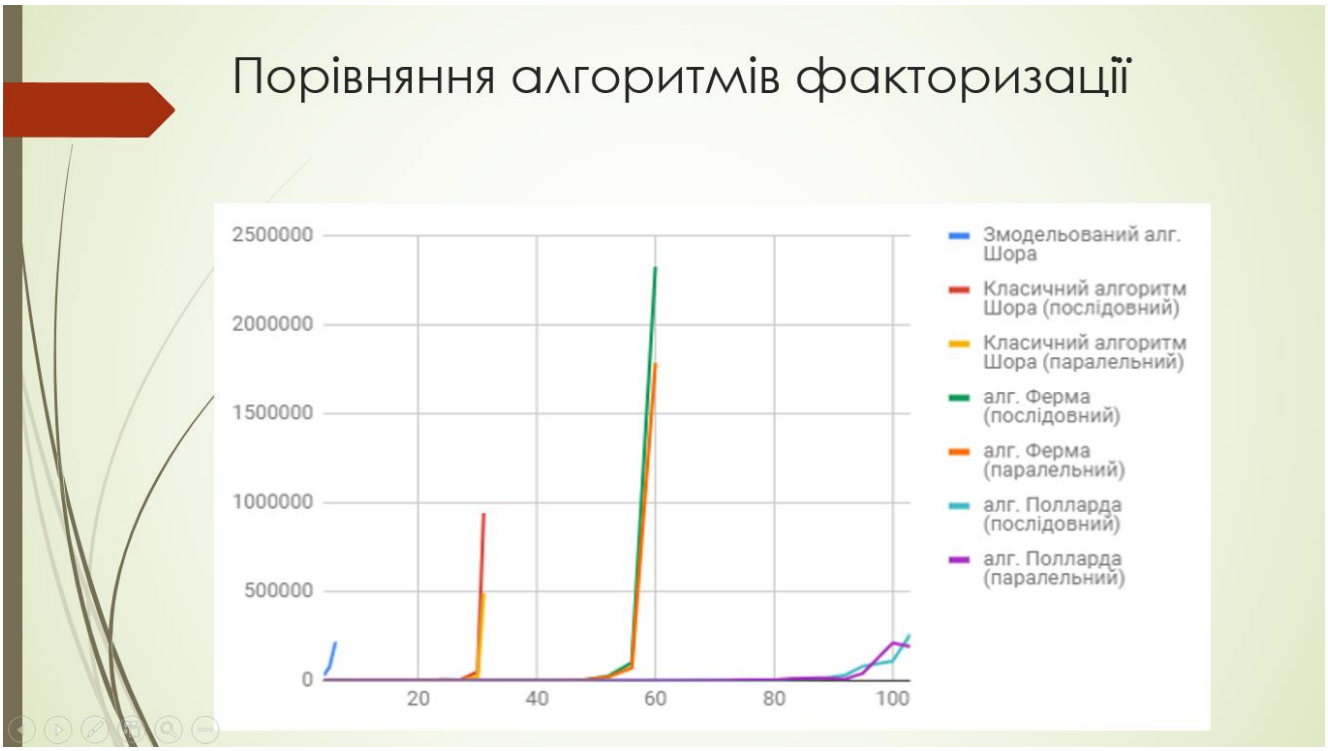

Рисунок А.14 – Слайд презентації №14

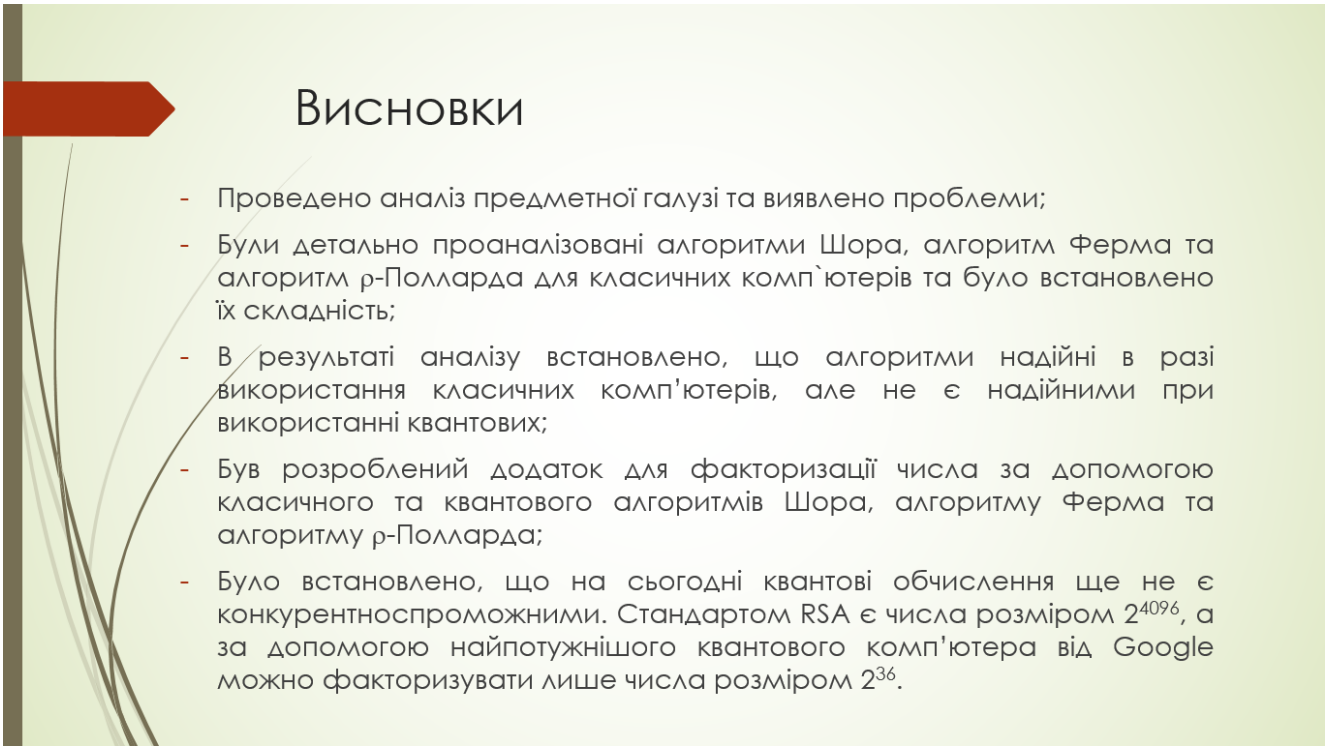

Рисунок А.15 – Слайд презентації №15

ДОДАТОК Б

Код реалізованих алгоритмів факторизації

Реалізація класичного алгоритма Шора

```
public class ClassicAlgorithm implements Algorithm {
    private final static BigInteger TWO = BigInteger.valueOf(2);
    private final static BigInteger ONE = BigInteger.ONE;
    private final static BigInteger ZERO = BigInteger.ZERO;

    @Override
    public Result factorize(BigInteger number) {
        long prime = 2;
        Result result;
        while (true) {
            result = factorize(prime, number);
            if (result.getResultStatus() == SUCCESS) {
                break;
            }
            prime = PrimeGenerator.getNextPrime(prime);
        }
        return result;
    }

    private Result factorize(long basis, final BigInteger
factorizedNumber) {
        Result result = new Result();
        result.setBasis(Long.toString(basis));
        BigInteger basisBI = BigInteger.valueOf(basis);

        // Check that basis is factor
        if (!basisBI.gcd(factorizedNumber).equals(ONE)) {
            result.setFirstFactor(basisBI.toString());

result.setSecondFactor(factorizedNumber.divide(basisBI).toString());
            result.setResultStatus(SUCCESS);
            return result;
        }

        for (BigInteger exponent =
BigInteger.valueOf(log(factorizedNumber, basisBI));
                exponent.compareTo(factorizedNumber) < 0;
                exponent = exponent.add(ONE)) {
            BigInteger moduleOfPoweredBasis =
basisBI.modPow(exponent, factorizedNumber);
            if (!moduleOfPoweredBasis.equals(ONE)) {
                continue;
            }
            else {
```

```
                if (!exponent.mod(TWO).equals(ZERO)) {
                    result.setResultStatus(INVALID_BASIS_PARAMETER);
                    return result;
                }
                BigInteger     basisInPeriod     =     pow(basisBI,
exponent.divide(TWO));
                if
(basisInPeriod.add(ONE).mod(factorizedNumber).equals(ZERO)) {
                    result.setResultStatus(INVALID_BASIS_PARAMETER);
                    return result;
                }

                BigInteger           firstVariableForGcd           =
basisInPeriod.subtract(ONE);
                BigInteger          secondVariableForGcd           =
basisInPeriod.add(ONE);

                BigInteger                  factor1                 =
firstVariableForGcd.gcd(factorizedNumber);
                BigInteger                  factor2                 =
secondVariableForGcd.gcd(factorizedNumber);

                result.setFirstFactor(factor1.toString());
                result.setSecondFactor(factor2.toString());
                result.setPeriod(exponent.toString());
                result.setResultStatus(SUCCESS);
                return result;
            }
        }
        result.setResultStatus(CANNOT_BE_FACTORIZED);
        return result;
    }
}
```

Реалізація алгоритма Ферма

```
public class FermatFactorization extends AbstractFactorization {
    @Override
    protected BigInteger innerFactorize(BigInteger n) {
        BigInteger x = BigIntegerSqrt.sqrt(n);
        BigInteger factorSqrt = x.multiply(x).subtract(n);
        BigInteger factor = null;

        while(!BigIntegerSqrt.sqrt(factorSqrt).pow(2).
                equals(factorSqrt)) {
            x = x.add(BigInteger.ONE);
            factorSqrt = x.multiply(x).subtract(n);
            BigInteger y = BigIntegerSqrt.sqrt(factorSqrt);
            factor = x.subtract(y);
        }
        if (factor == null) {
            BigInteger y = BigIntegerSqrt.sqrt(factorSqrt);
            factor = x.subtract(y);
```

```
        }
        return factor;
    }
}
```

Реалізація алгоритма ρ-Полларда

```
public class PollardFactorization extends AbstractFactorization {

    private final static BigInteger TWO = BigInteger.valueOf(2);
    private final static SecureRandom random = new SecureRandom();

    private BigInteger innerK;
    private BigInteger innerX;

    public PollardFactorization(int bitLength) {
        innerK = new BigInteger(bitLength, random);
        innerX = new BigInteger(bitLength, random);
    }
    @Override
    public BigInteger innerFactorize(BigInteger N) {
        final BigInteger K = innerK;
        BigInteger divisor;
        BigInteger X = innerX;
        BigInteger FX = X;
        if (N.mod(TWO).compareTo(BigInteger.ZERO) == 0) {
            return TWO;
        }
        do {
            X = X.multiply(X).mod(N).add(K).mod(N);
            FX = FX.multiply(FX).mod(N).add(K).mod(N);
            FX = FX.multiply(FX).mod(N).add(K).mod(N);
            divisor = X.subtract(FX).gcd(N);
        } while ((divisor.compareTo(BigInteger.ONE)) == 0);

        return divisor;
    }
}
```

Реалізація алгоритма Шора на мові програмування Q#

```
operation EstimatePeriod (a : Int, N : Int) : Int {
        EqualityFactB(IsCoprime(a, N), true, "`a` and `N` must be
co-prime");
        mutable result = 1;
        let bitsize = BitSizeI(N);
        let qubitsPrecision = 2 * bitsize + 1;
        repeat {
            mutable dyadicFractionNum = 0;
```

```
            using ((register1, register2)= (Qubit[qubitsPrecision]),
Qubit[qubitsPrecision]) {
                ApplyXorInPlace(1, register1);
                let oracle = DiscreteOracle(OrderFindingOracle(a, N,
_, _));
                oracle.apply(register1, register2);
                QFT(register1);
                let phase = M(register1);
                set dyadicFractionNum = Round(((phase *
IntAsDouble(2 ^ bitsPrecision)) / 2.0) / PI());
                ResetAll(register1);
                ResetAll(register2);
            }
            Message($"Estimated eigenvalue is
{dyadicFractionNum}/2^{bitsPrecision}.");
            let (numerator, period) =
(ContinuedFractionConvergentI(Fraction(dyadicFractionNum, 2 ^
bitsPrecision), N))!;
            let (numeratorAbs, periodAbs) = (AbsI(numerator),
AbsI(period));
            Message($"Estimated divisor of period is {periodAbs}, "
+ $" we have projected on eigenstate marked by {numeratorAbs}.");
            set result = (periodAbs * result) /
GreatestCommonDivisorI(result, periodAbs);
        }
        until (ExpModI(a, result, N) == 1)
        fixup {
            Message("It looks like the period has divisors and we
have " + "found only a divisor of the period. Trying again ...");
        }
        return result;
    }
```

ДОДАТОК В

Підготовлені тези на 23-го міжнародний молодіжний форум "Радіоелектроніка та молодь у XXI столітті"

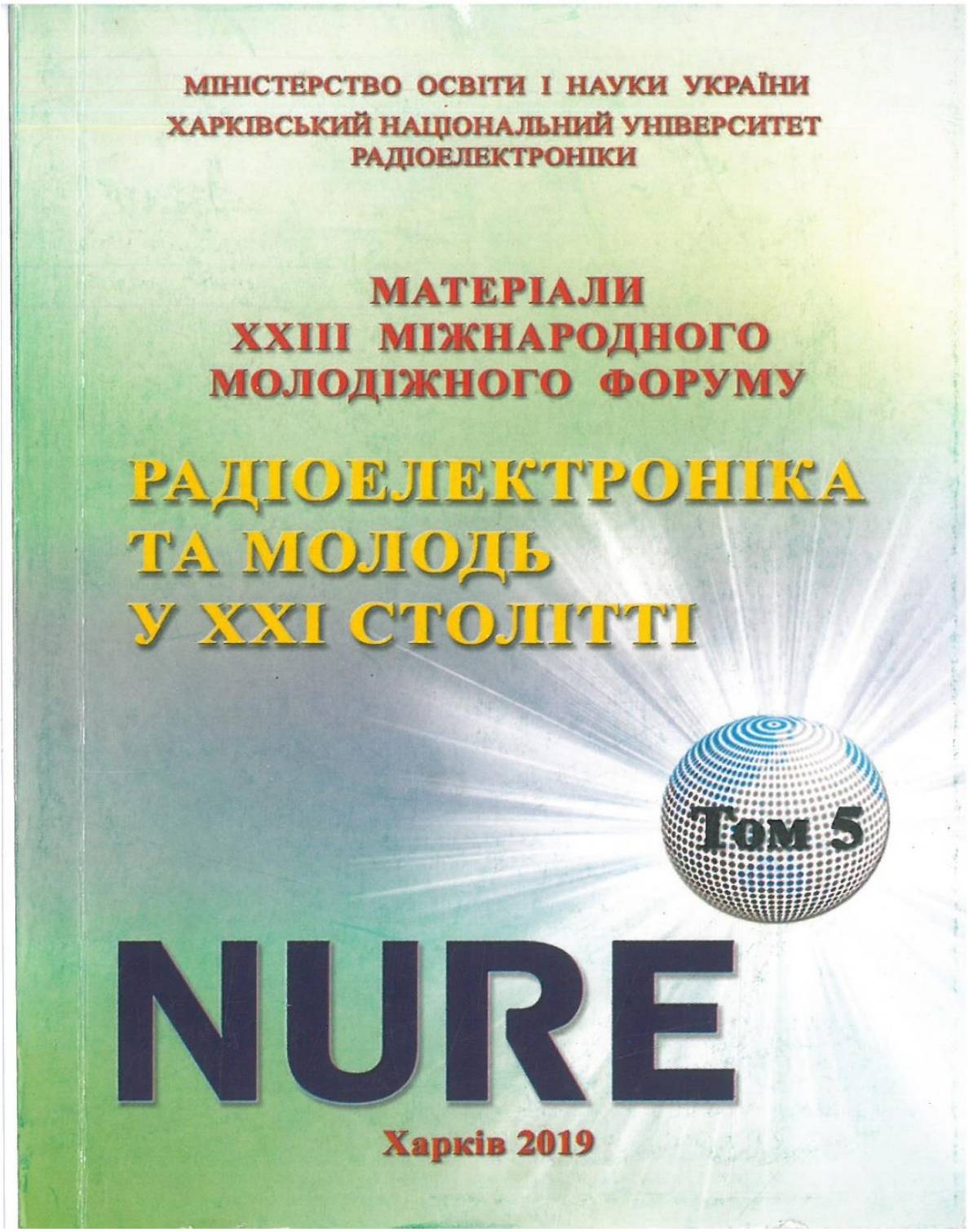

Рисунок В.1 – Титульна сторінка збірника

КВАНТОВИЙ АЛГОРИТМ ШОРА ДЛЯ ВИРІШЕННЯ ПРОБЛЕМИ ФАКТОРИЗАЦІЇ ЧИСЛА

Островський А.М.
Науковий керівник – к.т.н., проф. Качко О.Г.
Харківський національний університет радіоелектроніки
(61166, Харків, просп. Науки,14,
каф. Системотехніки, тел. (057) 702-13-06)
e-mail: albert.ostrovskyi@nure.ua, факс (099) 728-75-48

The given work is devoted to the modern developments in the field of quantum computer algorithms related to cryptanalysis. The major topic of the work is a research of the Shor quantum algorithm that is designed to solve a factorization problem. The work also contains information about the current state of quantum computing technologies and known non-quantum solutions to the factorization problem. The aim of the work is to demonstrate that nowadays-cryptographic systems based on impossibility to find factors of the big number are vulnerable to quantum algorithms or will be vulnerable in the near future.

Несиметричні алгоритми шифрування широко використовуються у сучасних криптосистемах. Найвідомішим є алгоритм RSA, стійкість якого базується на складності задачі факторизації. Для забезпечення безпеки, RSA вимагає, щоб добуток випадкових простих чисел був більше ніж 300 десяткових цифр. Навіть при використанні найшвидшого комп'ютера, доступного на сьогодні, розкладання на множники цілого числа такого розміру вимагає нездійсненно великого часу. Це означає, що RSA безпечний, поки не буде знайдений ефективний алгоритм розкладання на множники.

Досліджені алгоритми розкладання числа на множники для звичайних ЕОМ. Найкращі мають субекспоненціальні складність, це метод квадратичного решета (складність $L_n(0.5,1)$) та метод Ленстра з використанням еліптичних кривих (складність $L_n(0.5,\sqrt{2})$), де L_n це нотація прийнята для позначення субекспоненціальної складності. Тобто жоден з сучасних алгоритмів не може знайти співмножники великого цілого числа з поліноміальною складністю часу [1].

Найвірогідніше рішення цієї проблеми є використання квантових обчислень. Квантова система з L дворівневих кубітів має 2^L лінійно незалежних станів, тобто квантовий обчислювальний пристрій розміром L кубіт може виконувати паралельно 2^L операцій [2]. Компанією Google вже реалізований квантовий процесор з 72 кубітів.

Вивчено квантовий алгоритм Шора для факторизації (розкладання числа на прості множники), реалізація якого, дозволяє розкласти число M за час $O(\lg^3 M)$, використовуючи $O(\lg M)$ логічних кубітів. Сутність

Рисунок В.2 – Сторінка збірника 89

алгоритму - знайти період функції $f(x) = a^x (mod\, M)$, де a довільний параметр, M число, яке необхідно факторизувати. Знайшовши період функції, можемо знайти множники числа за допомогою формули $factor_{1,2} = \gcd(a^{r/2} \pm 1, M)$, де $\text{factor}_{1,2}$ – множники числа M, gcd – формула для обчислення найбільшого спільного дільника, r – період. Парадигма квантових обчислень дозволяє знайти період функції з поліноміальною складністю, чого не можна зробити в рамках класичної обчислювальної моделі [3].

Обрана обчислювальна схема для реалізації квантової частини алгоритму, яка представляє собою два квантових регістра m та n, які спочатку знаходяться у нульовому стані. На першому кроці за допомогою операції Уолша-Адамара первинний стан $|0\rangle$ регістра n переводиться в рівно імовірнісну суперпозицію усіх булєвих станів N. На другому кроці до двух регістрів застосовується унітарне перетворення (оракул S_f), яке переводить стан $|x, 0\rangle$ у $|x, f(x)\rangle$. На третьому кроці виконується квантове перетворення Фур'є (QFT), яке уявляє собою унітарне перетворення стану квантового регістра, яке задається N-мірним вектором стану. В результаті ми отримуємо $\frac{1}{N}\sum_{x=0}^{N-1}\sum_{k=0}^{N-1} \exp(2\pi\, i\, kx/N)\, |k, a^x\, mod\, M\rangle$, що є перетвореним першим регістром, де N – кількість усіх можливих булєвих станів. На четвертому кроці виконується вимірювання регістра X відносно ортогональної проекції. Результатом є $|k, a^k\, mod\, M\rangle$, де $k \in [0, N-1]$. Далі знаходимо найкраще приближення (знизу) до k/N, що і буде періодом функції, який потім використовується для знаходження множників числа за допомогою класичних обчислень. Також треба зазначити, що алгоритм є ймовірнісним, де ймовірність знаходження відповіді є $\frac{1}{N}\sum_{x:a^x \equiv a\, mod\, M}^{N-1} \exp(2\pi\, i\, kx/N)$ [4].

Алгоритм є теоретично обґрунтованим та вже використовувався для факторизації числа 15 на справжньому квантовому комп'ютері. Для подальшого дослідження алгоритму буде виконана його реалізація за допомогою однієї з мов програмування для квантових обчислень (Q# та QCL) та аналіз його роботи на більш великих числах.

Список джерел:

1. Ишмухаметов Ш.Т., Методы факторизации натуральных чисел [Текст]: учеб. пособие / Ш.Т. Ишмухаметов.– Казань:Казан.ун., 2011.-201с.

2. Роман Душкин Квантовые вычисления и функциональное программирование / Р.В. Душкин.-М.: ДМК Пресс, 2015. – 232 с.

3. К.А. Валиев Квантовые компьюетры и квантовые вычесления/К.А. Валиев.-М.:Insitute of Physics and Technology, 2005.- 37с.

4. Shor P. Algorithms for quantum computation [Text]/Shor P.// Foundations of Computer Science.–1994.–№10.-124–134pp.

Рисунок В.3 – Сторінка збірника 90